Mu #2

04#2
7-13-01

169.2089

PATENT APPLICATION

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of:	)	
	:	Examiner: N.Y.A.
JOHN CHARLES BROOK	)	
	:	Group Art Unit: N.Y.A.
Application No.: 09/893,645	)	
	:	
Filed: June 29, 2001	)	
	:	
For: HASH COMPACT XML PARSER	)	July 27, 2001

Commissioner for Patents
Washington, D.C. 20231

CLAIM TO PRIORITY

Sir:

Applicant hereby claims priority under the International Convention and all rights to which he is entitled under 35 U.S.C. § 119 based upon the following Australian Priority Application:

PQ8495, filed June 30, 2000.

A certified copy of the priority document is enclosed.

Applicant's undersigned attorney may be reached in our New York office by telephone at (212) 218-2100. All correspondence should continue to be directed to our address given below.

Respectfully submitted,

[illegible]
Attorney for Applicant

Registration No. 29,286

FITZPATRICK, CELLA, HARPER & SCINTO
30 Rockefeller Plaza
New York, New York 10112-3801
Facsimile: (212) 218-2200

NY_MAIN187928v1

CFP/169.2089
09/893,645

AUSTRALIA

OIPE JC154
JUL 3 1 2001
PATENT & TRADEMARK OFFICE

Patent Office
Canberra

I, CASSANDRA RICHARDS, TEAM LEADER EXAMINATION SUPPORT AND SALES hereby certify that annexed is a true copy of the Provisional specification in connection with Application No. PQ 8495 for a patent by CANON KABUSHIKI KAISHA filed on 30 June 2000.

I further certify that pursuant to the provisions of Section 38(1) of the Patents Act 1990 Application No. 54113/01 is associated with Provisional Application No. PQ 8495 filed on 30 June 2000.

COMMONWEALTH OF AUSTRALIA
AUSTRALIA
PATENT OFFICE

WITNESS my hand this
Tenth day of July 2001

[illegible]

CASSANDRA RICHARDS
TEAM LEADER EXAMINATION
SUPPORT AND SALES

CERTIFIED COPY OF
PRIORITY DOCUMENT

S&F Ref: 493128

ORIGINAL

AUSTRALIA

Patents Act 1990

PROVISIONAL SPECIFICATION FOR THE INVENTION ENTITLED:

Hash Compact XML Parser

Name and Address of Applicant:

Canon Kabushiki Kaisha, incorporated in Japan, of 30-2, Shimomaruko 3-chome, Ohta-ku, Tokyo, 146, Japan

Name of Inventor:

John Charles Brook

This invention is best described in the following statement:

HASH COMPACT XML PARSER

Copyright Notice

Technical Field of the Invention

The present invention relates generally to processing of multimedia documents, and, in particular, to documents produced in mark-up language. The present invention relates to a method and apparatus for parsing documents in mark-up language. The invention also relates to a computer program product including a computer readable medium having recorded thereon a computer program for parsing a document composed in a mark-up language.

Background Art

Parsing is a process of extracting information from a document. The process usually involves at least a minimum check of document syntax, and can in general yield either a tree structure description of the document, or a logical chain of events. The structural representation based on the logical chain of events is typically produced by an ordered parsing of the document from beginning to end.

Tree based parsers compile an XML document into an internal tree structure, providing a hierarchical model which applications are able to navigate. The Document Object Model (DOM) working group at the World-Wide Web consortium is presently developing a standard tree-based Application Programming Interface (API) for Extended Markup Language (XML) documents. Event based parsers on the other hand, report parsing events such as the start and end of elements directly to the application for which the parsing is being performed. This reporting is performed typically using callbacks, and does not require an internal tree structure. The application requiring the parsing implements handlers to deal with the different events, much like handling events in a graphical user interface.

Tree based parsers are useful for a wide range of applications, but typically place a strain on system resources, particularly if the document is large. Furthermore, applications sometimes need to build their own particular tree structures, and it is inefficient to build a tree representation, only to map it to a different representation. Event based parsers provide a simpler, lower-level access to an XML document, facilitating parsing of documents larger than available system memory. The "Simple API for XML" (ie. SAX) parser is an event-driven interface for parsing XML. SAX parsers are discussed in more detail in relation to Figs. 2 and 3.

Fig. 1 shows a block representation of a prior art XML parser system. In the figure, the following XML document fragment 106 is provided:

```
105  <Shakespeare>
110  <!--This is a comment-->
115  <div class="preface" Name1="value1" name2="value2">
120  <mult list=&lt;> </mult>
125  <banquo>
130  Say                                                          [1]
135  <quote>
140   goodnight   </quote>,
145  Hamlet.</banquo>
150  <Hamlet><quote>Goodnight, Hamlet. </quote></Hamlet>
155  </Shakespeare>
```

In the lower portion of the figure, the XML document 106 is input into a parser 112 which, in the present instance, is an event based parser. Optionally, as indicated by a dashed box 108, a Document Type Definition (DTD) is also input into the parser 112. The parser 112 outputs a partial structural representation of the document 106 which can be a simple list. In the upper portion of the figure, a Cascading Style Sheet (CSS) or an Extendable Style Sheet (XSL) 104 is input into a CSS or XSL parser 110. A DTD 102 can also be input into this parser 110. Both the XML parser 112 and the CSS/XSL parser 110 are event driven parsers in the present illustration.

One of the benefits of mark-up languages such as XML, is the facility to make documents smarter, more portable and more powerful, by enabling the use of tags to define various parts of the documents. This capability derives from the descriptive nature of XML. XML documents can be customised on a per-subject basis, and accordingly, customised tags can be used to make the documents comprehensible, in terms of the structure, to a human reader. This very attribute, however, often leads to XML documents being verbose and large, and this poses a problem in some instances. For example, where XML documents must be parsed in a hardware-constrained piece of equipment, such as a printer, the typically memory intensive nature of conventional parsing is in conflict with the limited memory which can be accommodated in such equipment. Furthermore, tag-string matching operations, which are required to a significant degree in XML document parsing, pose a sometimes unacceptable burden of processing requirements, translating into an unacceptable number of processor cycles. These problems apply to both parser instances shown in Fig. 1.

Disclosure of the Invention

It is an object of the present invention to substantially overcome, or at least ameliorate, one or more disadvantages of existing arrangements.

According to a first aspect of the invention, there is provided a method of parsing a markup language document comprising syntactic elements, said method comprising, for one of said syntactic elements, the steps of:

identifying a type of the element;

processing the element by determining a hash representation thereof if said type is a first type; and

augmenting an at least partial structural representation of the document using the hash representation if said type is said first type.

According to another aspect of the invention, there is provided a method of encoding a markup language document comprising syntactic elements, said method comprising, for one of said syntactic elements, steps of:

identifying a type of the syntactic element; and

processing the syntactic element by one of:

(i) determining a hash representation thereof if said type is a first type;

(ii) determining a compressed representation thereof if said type is not a first type; and

(iii) retaining the syntactic element.

According to another aspect of the invention there is provided a method of decoding a markup language document comprising encoded syntactic elements, said method comprising, for one of said encoded syntactic elements, steps of:

identifying a type of the encoded syntactic element;

processing the encoded syntactic element by at least one of:

(i) determining an inverse hash representation thereof if said type is a first type; and

(ii) determining a decompressed representation thereof if said type is not a first type; and

(iii) retaining the encoded syntactic element.

According to another aspect of the invention there is provided a apparatus for parsing a markup language document to form an at least partial structural representation of the document, said apparatus adapted to perform any one of aforementioned methods.

According to another aspect of the invention there is provided a apparatus for one of encoding and decoding a markup language document to form an at least partial structural representation of the document, said apparatus adapted to perform any one of aforementioned methods.

According to another aspect of the invention there is provided a computer program product including a computer readable medium having recorded thereon a computer program for implementing an apparatus for parsing a markup language document to form an at least partial structural representation thereof, said program adapted to perform any one of the aforementioned methods.

Brief Description of the Drawings

A number of preferred embodiments of the present invention will now be described with reference to the drawings, in which:

Fig. 1 shows block representations of XML parser systems in which embodiments of the present invention can be practiced;

Fig. 2 depicts a flow chart of method steps for a prior art SAX parser, including optional well-formedness and/or validation checking steps;

Fig. 3 shows the SAX parser of Fig. 2 in accordance with a preferred embodiment of the present invention;

Fig. 4 is a schematic block diagram of a special purpose embedded computer upon which an embodiment of the present invention can be practiced; and

Fig. 5 is a general purpose computer upon which an embodiment of the present computer can be practiced.

Detailed Description including Best Mode

Where reference is made in any one or more of the accompanying drawings to steps and/or features, which have the same reference numerals, those steps and/or features have for the purposes of this description the same function(s) or operation(s), unless the contrary intention appears.

In the context of this specification, the word "comprising" means "including principally but not necessarily solely" or "having" or "including" and not "consisting only of". Variations of the word comprising, such as "comprise" and "comprises" have corresponding meanings.

The inventive concept disclosed in this specification is based on the idea that memory requirements of an XML parser can be reduced, and various performance metrics can be improved, by performing a "perfect" hash of the XML tags, and possibly other elements within an XML file. A hash function is a function, mathematical or otherwise, that takes an input string, and converts it to an output code number called a hash value. A perfect hash function is one which creates a unique code number for a unique input string within a preset domain. The input string can be composed, for example, of alpha numeric characters, or other characters approved by the World Wide Web Consortium, and must be less than a certain length dictated by the hash process specifics. Alternatively, or in addition, the input string can be constrained in other ways, for example in terms of a probability of code number collision based on input context. This idea allows any arbitrary XML tag to be treated as a numeral or code, and stored in numeric form in memory. Since a parser must normally preserve some portion of an XML structure in memory as it is parsed, conversion of XML tags to unique numerals allows memory requirements to be reduced, and furthermore, allows string-to-string comparisons to be replaced with much faster numerical comparisons.

The principles of the preferred method described herein have general applicability to parsing a wide variety of mark-up languages. For ease of explanation, the steps of the preferred method are described with reference to the XML language. It is not intended, however, that the present invention be limited to the described method. For example, the invention can also be applied to a UTF-16 transformation format (see International Standard ISO/IEC 10646-1 for further details of UTF-16).

Fig. 2 depicts a prior art SAX parser process 236, which supports optional well-formedness and/or validation checking sub-processes. A mark-up document, in the present case an XML document, is opened in an initial sub-process 200. A decision sub-process 202 tests whether the document contains any unprocessed (ie unparsed) characters, and if this is the case, a character is read and stored in a string in a sub-process 204. If further characters are not detected in the testing sub-process 202, the parsing processing terminates in a sub-process 234.

In a following testing step 206, a check is performed as to whether a complete syntactic element has yet been assembled, and if so, the parser process 236 proceeds to a "Syntactic Type" identification step 210. If, on the other hand, a complete syntactic element has not yet been assembled, the parser process 236 is directed to a decision block 208 which tests if any further characters are available in the document. If additional characters are available, the parser process 236 is directed according to a "yes" arrow back to the step 204. Alternatively, if no more characters are available, then the process 236 is directed in accordance with a "no" arrow to a syntactic element type identification step 210.

The "type identification" step 210 identifies a "type" for the assembled syntactic element, after which the element string is placed, in a step 212, into a memory representation of the document structure, as assembled to this point. The memory representation of the document structure, which is typically, in the case of event driven parsers, a partial structural representation of the document, can be a simple list.

Thereafter, in an optional sub-process 238, a "well-formedness" check can be performed in a sub-process 214. The optional nature of step 238 is denoted by the dashed block outline of the step and of the component steps thereof. If the optional sub-process 238 is not elected, the parsing process 236 can be directed to an optional sub-process 240, or alternately, to an action selection step 226.

If the optional sub-process 238 has been elected, then after the well-formedness step 214, if an error is detected in the following error checking step 216, corrective action and/or error indication takes place as indicated by an arrow 218. If, on the other hand, no errors are detected, the parser process 236 can be directed to an optional sub-process 240, in which a validation check can be performed in a sub-process 220.

The "validation check" sub-process 220 involves a comparison of the identified syntactic element against a document type definition (DTD). This comparison procedure verifies correct syntactic placement of elements to a greater extent than the mere well-formedness check. The optional nature of the step 240 is denoted by a dashed block outline of the step 240 and of the component steps thereof.

Subsequently, if an error is detected in an error checking step 222, corrective action is taken, and/or an error indication is produced, as depicted by an arrow 224. Alternatively, if no error is detected, the parser process proceeds to a selection sub-process 226, where an action is selected based upon the type of the syntactic element being considered.

If the syntactic element is a tag, then its value, or a representative string, is sent to the application in respect of which the parsing process is being performed, and a memory representation of the tag is maintained, this being depicted by an arrow 228. If, on the other hand, the element type is a non-tag type, its value string is sent to the associated application, and the memory representation of the element is deleted. Finally, the parsing process 236 is directed, as depicted by an arrow 232, back to the character testing step 202.

The memory requirements arising from the verbose nature of the XML document can be discerned in the memory requirements associated with the memory representation of the document structure in its original string form, this being referred to in the sub-process 212. Furthermore, an associated processing load, relating to the need to perform string comparisons between variable length alpha-numeric strings, arises both in the well-formedness checking sub-process 214, and in the validation checking sub-process 220.

Reasons for preserving a partial memory representation of the document, and for performing string checking, include (i) in relation to the sub-process 214; checking for closure of hierarchy branches ie matching end tags to start tags, and checking for non-overlapping branches, and (ii) in relation to the sub-process 220; similar objectives as in (i), as well as checking conformity of structure and tag names against the DTD.

A parser must normally preserve some portion of an XML structure in memory as it is parsed. Even for a SAX parser, a local portion of the XML structure must be retained in memory for correct operation. If however each XML tag is converted to a unique numeral using a hash function, memory requirements are typically reduced, since the numeral resulting from the hash operation is smaller than the associated arbitrary-length XML tag string. Furthermore, string-to-string comparisons, required for matching beginning & end tags, can be replaced with much faster numerical comparisons.

Typical hash algorithms include (i) Cyclic Redundancy Coding (CRC) algorithms (commonly used for signature analysis or error-detection/correction in data transfer & storage), (ii) fully lossless encoding algorithms, and (iii) Huffman encoding algorithms.

Typically, a suitable hash algorithm must be static in its operation, that is, it must always return the same hash result for the identical input conditions over the required set of data. The required set of data can, however, vary with circumstance. The data set can thus typically comprise at least an entire markup document, but can also include a relevant DTD, linked markup documents, and related or linked markup documents in different languages, eg a CSS document referenced by an XML document. A static hash algorithm can, however, be achieved where necessary by resetting the algorithm whenever tag syntax is encountered, eg whenever the non-literal '<' character is found in an XML document. The hash algorithm can also be reset where an <!ELEMEMT string is found in an XML DTD document, or wherever a valid tag selector is permitted in a CSS document.

Any suitable hash algorithm to be used within an embodiment can be signalled or selected by a reference in any input markup document. This can be done in much the same way as markup documents can reference other markup documents, DTDs, stylesheets, character encodings, namespaces, and so on. For example a particular hash algorithm can be identified with a particular namespace, thereby permitting indirect reference to a hash algorithm via a namespace reference within a document. A hash algorithm implementation can be wholly, or partially included within a markup document, along with associated parameterisation. Such methods of referencing or including hash algorithms can be useful for optimisation purposes, where different hash methods have been optimised for particular markup documents, thereby improving performance and memory usage in destination devices or systems. Alternatively, the aforementioned referencing methods can be useful for matching purposes. This refers to applications involving one or more markup documents, where error checking or completion of parsing or other functions are required, and where one or more other documents (e.g. a DTD) have already been hashed by the same algorithm.

Further extensions are possible to the above method, for example optional hashing of DTDs. This reduces Read Only Memory (ROM) requirements for storing

DTDs, and provides for faster validation processing of XML documents, by allowing comparison of numerical values rather than (slower) string comparisons.

Fig. 3 illustrates a SAX parser process 344 now in accordance with a preferred embodiment of the present invention. Sub-processes 300, to 310 are identical to corresponding sub-processes 200 to 210 which were described in relation to Fig. 2. Thereafter, the assembled syntactic element is tested for its nature as a tag, or otherwise, in a testing sub-process 312. If the element is a tag, the parsing process 344 is directed to a hash sub-process 318 by an arrow 316. The hash step 318, determines a unique numeric representation of the element. This is a more memory efficient representation of the element, and also lends itself to simpler and faster comparison operations in the numeric, rather than the alpha-numeric domain. Both the element string, and the hash value thereof, are retained at this point, however it is the hash value which is inserted, in the step 318, into the memory representation of the document structure, and not the string value.

In order to appreciate the parsing process 344 as described in relation to Fig. 3 to this point, parsing of the XML fragment [1] is considered firstly in relation to the parsing process 236 described in relation to Fig. 2. In this case, the XML fragment [1] yields the following hierarchical representation of parsed mark-up tags in the sub-process 212:

```
205  Shakespeare
215         div
220          mult
221          /mult
225         banquo
235                 quote
240                quote                                                  [2]
245         /banquo
250          Hamlet
251                   quote
252                  /quote
253           /Hamlet
255  /Shakespeare
```

In contrast, the differentiated treatment of tag elements and non-tag elements in the parsing process 344 as described in relation to Fig. 3 results in a hierarchical representation of the equivalent hashed sub-process 318, as depicted in [3]. The hierarchical representation in [3] is made up of parsed hashed mark-up tags. For the sake of this example, a domain of tag names is constrained to those shown in the following Table 1, and a hash mapping (which is functionally equivalent to a hash "function") is shown in the following table:

Tag	Hash Code Number
Shakespeare	133
div	326
mult	371
banquo	787
quote	629
Hamlet	411

Table 1. Hash Mapping

Based on the above hash mapping, the following representation results:

```
205  133
215          326
222            371
223            /371
225          787
235                    629
240                   /629                                        [3]
245          /.787
254            411
255                        629
256                       /629
257            /411
255  /133
```

Continuing with the description of Fig. 3, the parsing process 344 can be directed to an optional step 346, in which a well-formedness sub-process 320 can be performed. The optional nature of the process 346 and the sub-steps thereof, is depicted by use of dashed lines. If the optional sub-process 346 is not elected, then the parsing process 344 can be directed to an optional sub-process 348 where validation checking is performed. Alternately, the parsing process 344 can be directed to an action selection step 334.

It is apparent that processing associated with the hierarchical representation depicted in [3] allows string comparisons to be replaced by faster and more efficient numerical comparisons, thereby reducing an associated computational burden. Furthermore, the hierarchical representation shown in [3] is a more memory-efficient representation, and accordingly more suited to memory-constrained applications as previously discussed.

If the optional sub-process 346 has been elected, then after well-formedness checking is performed in the step 320, the parsing process 344 is directed to an error checking step 322, whereupon if an error is detected, corrective action is taken, and/or an error is indicated as depicted by an arrow 324. If, on the other hand, no error is detected, the parsing process 344 can be directed to the optional step 348, in which a validation checking step 326 can be performed. The optional nature of the step 348 is again indicated by the dashed lines. In order to perform the validation step 326, DTD tags are first hashed in a hashing step 328, in order to bring the DTD memory representation into conformity with the hashed nature of the mark-up document, as generated by the hash sub-process 328. The validation checking step 326 searches the memory representation of the DTD structure for verification of correct syntactic placement of elements, noting that string comparisons as used in step 220 in relation to Fig. 2 have now been replaced by faster and more efficient numerical comparisons as a result of the hashing operation in step 328.

After validation, an error checking step 330 is performed, with corrective action and/or error indication being performed as indicated by an arrow 332. If no errors are detected, the parsing process 344 proceeds to an action selection step 334, whereupon if the syntactic element is a tag type, the tag string is sent to the application in respect of which the parsing process is being performed, and the tag string is deleted from memory. The associated hashed tag memory representation is, however, retained. Accordingly, no string-based memory representation of the tag is retained, other than one copy of the currently parsed tag string. The memory representation of the tag is thus only in hashed form. Continuing with the description, if the element syntactic type is a non-tag type, its value or string is sent to the associated application, and the associated memory representation is deleted. The parsing process now loops back, as indicated by an arrow 340, to the character testing step 302. If no further characters are detected, the parsing process terminates in a sub-process 342.

The XML document fragment [1], with tags in hashed form, has the following form:

```
505 <133>
110 <!--This is a comment-->
515 <326 class="preface" Name1="value1" name2="value2">
520 <371 list=&lt;> </371>
525 <787>
130 Say                                                        [4]
535 <629>
540  goodnight   </629>,
545 Hamlet.</787>
550 <411><629>Goodnight, Hamlet. </629></411>
555 </133>
```

The representation of closing tags (i.e. syntax: </section> as opposed to start tags <section>) can be defined in various ways, attaining more, or less, compatibility with the XML standard. This compatibility can be more important in some instances than in others. In the preferred embodiment, the '/' character of a current tag string is typically removed prior to hashing the following tag string, in order that identical start and end tags return the same numeral from the hashing function. Alternatively, there may be situations where it is desired to retain an equivalent representation of the '/' character (identifying the end tag) in memory. This can be done in a variety of ways, such as: (i) reinserting the '/' character or an equivalent character into memory in proximity to the end tag hash numeral so as to indicate that it is an end tag (ii) using a boolean value to indicate the end or start state of a hashed tag (iii) negating the end tag hashed value so that a simple addition of start and end tags yields zero for a perfect match. Option (iii) requires that a sign bit be guaranteed free from influence by the hashing algorithm, and is in fact very similar to the boolean flag option (ii).

Furthermore, structured hash numbers can be generated, in which a hash number for a nested tag can indicate the higher-level XML tags in which the first tag is nested (e.g. instead of nested tag 123, label it as 987.123, where tag 123 is nested inside tag 987). This structured hashing can allow further parsing performance improvements by reducing structure-spanning operations, ie by reducing an amount of the XML document which must be held in memory while the end-points of a tag pair are being searched for.

A structured equivalent hashed markup example for the XML fragment [3] is presented in [5] below using negated, hashed end tags.

```
133.326
133.371
133.-371
133.787.629          ->  013307870629
133.787.-629   -> -013307870629
133.-787                                                    [5]
133.411
133.411.629
133.411.-629
133.-411
-133
```

In [5], the structure of nested tags is converted from the form shown in [3] into a series of linked hashed tags in which each subsequent lower level of hashed tag can be directly linked to its upper levels. This allows simple numerical comparison with a similarly parsed structure from a hashed DTD. In fact, each line in [5] can be represented, as shown in [5] for lines 4 and 5, by a single numeral which is combined from the set of hashed tags encountered. This single numeral can represent in a very compressed form both the identities and relationships of the original input tags and can accordingly provide a very efficient comparison method with a similarly hashed DTD. A single or multiple set of numerical comparisons between a tag set from the parsed & hashed input document and a line from the parsed & hashed DTD replaces a series of string and structure comparisons normally encountered in XML parser validation.

An "imperfect" hash process ie a hash process not guaranteed to produce a unique numeral for each alphanumeric input string, can be adequate in certain cases, in particular where the maximum length of XML tag strings is constrained, or constrained to some probability. Furthermore, in cases where the set of XML tag strings is constrained to some limited number of character permutations, or constrained with some probability to a limited number of character permutations, the imperfect hash process can be designed, or selected, to operate adequately.

A communications standard or alternative public or private format(s) can be defined or described based on the use of a hash algorithm. This technique allows a form of compression, which can be of benefit to transmission of XML data because of its typical verbosity and human-readable, ASCII form. Various options exist for retaining or discarding human-readability, for example by combining (perfect) hashing with other forms of compression (applied to differing element types within an XML file). For instance, it is possible to replace XML string tags with unique, human-readable numerals derived from a perfect hashing algorithm. Un-hashed syntactic and other elements can also be compressed by a lossless compression technique for transmission between processes or devices, thereby reducing the amount of transmitted data.

An inverse or reversible hash algorithm can be referenced or included where required as previously discussed. This can arise where, for example, such an algorithm is needed to decode or decrypt one or more markup tags into a human-readable string for display or labelling purposes from a pre-hashed, transmitted markup document where it is otherwise not necessary to do this for parsing and error-checking purposes. Another possible use of a reverse or inverse hash algorithm is to allow decryption of markup tags or other data to allow enabling of some restricted function or feature relating to the transmitted markup document. This method can also be used for matching of a transmitter and a receiver of markup documents, where the reverse or inverse hash algorithm is already included in the receiver (and is not transmitted but might be referenced in the markup document). Examples of reversible or invertible hash algorithms include (i) fully lossless encoding algorithms and (ii) Huffman encoding algorithms.

The aforementioned embodiments can be applied to any markup language, with particular advantages where one or more of the following conditions apply, namely (i) the markup language allows definition of tag names (e.g. XML, DTD, CSS, XSL, etc) (ii) tag names use large character encoding tables (e.g. UTF-16) and/or tag name length is not short (iii) the intended application using or receiving a markup document typically requires representation of complex structures with more than one level of nesting within a markup document or DTD (iv) some form of checking (typically well-formedness or validation) is required of the input markup document (v) the markup parser and/or application have strong limitations on memory capacity (e.g. embedded or low-cost CPU system) or memory management (e.g. no virtual memory or no dynamic memory allocation), and (vi) the markup parser and/or application have a requirement to operate quickly on complex (highly-nested) markup documents.

The method of parsing a markup language document can be implemented in dedicated hardware such as one or more integrated circuits performing the functions or sub functions of parsing a markup language document. Such dedicated hardware may include graphic processors, digital signal processors, or one or more microprocessors and associated memories.

The method of parsing a markup language document can alternatively be practiced using a special purpose embedded computer system 400, such as that shown in Fig. 4 wherein the process of Fig. 3 may be implemented as software, such as an application program executing within the embedded computer system 400. The computer system 400 is typically integrated (ie embedded) into an end system such as a printer (not shown) and drives a printer engine 402 in the printer. In particular, the steps of the method of parsing a markup language are effected by instructions in the software that are carried out by the embedded computer. The software may be stored in a computer readable medium, including Read Only Memory (ROM), Random Access Memory (RAM) or other types of memory. The software is loaded into the embedded computer during manufacture, or by software upgrades performed on-site.

The embedded computer system 400 comprises a computer module 410, input devices such as a switch module 422 for parameter setting, an output device such as a Liquid Crystal Display (LCD) showing job status, and the printer engine 402. The embedded computer 400 is typically physically integrated into the printer (not shown). Print jobs which originate at other computers attached to a computer network 406 are sent to the embedded computer 400 by a connection 404 to an Input/Output (I/O) interface 408.

The embedded computer module typically includes a processor unit 414, a memory unit 418, for example formed from semiconductor random access memory (RAM) and read only memory (ROM), input/output (I/O) interfaces including a switch module and LCD interface 416, and an I/O interface 408 for the printer engine 402 and network 406. The components 408, and 414 to 418 of the embedded computer 410 typically communicate via an interconnected bus 412 and in a manner which results in a conventional mode of operation of the embedded computer system 410 known to those in the relevant art. Typically, the program of the embodiment is resident in memory 418, and is read and controlled in its execution by the processor 414.

The method of parsing a markup language document can also be practiced using a conventional general-purpose computer system 500, such as that shown in Fig. 5 wherein the process of Fig. 3 may be implemented as software, such as an application program executing within the computer system 500. This application is useful, for example, when hashing is used as a communication standard across a network between computers. Fig. 5 shows only one of the communicating computers being considered.

In particular, the steps of the method of parsing a markup language document are effected by instructions in the software that are carried out by the computer. The software may be divided into two separate parts, namely one part for carrying out the parsing methods, and another part to manage the user interface between the latter and the user. The software may be stored in a computer readable medium, including the storage devices described below, for example. The software is loaded into the computer from the computer readable medium, and then executed by the computer. A computer readable medium having such software or computer program recorded on it is a computer program product. The use of the computer program product in the computer preferably effects an advantageous apparatus for parsing a markup language document in accordance with the embodiments of the invention.

The computer system 500 comprises a computer module 501, input devices such as a keyboard 502 and mouse 503, output devices including a printer 515 and a display device 514. A Modulator-Demodulator (Modem) transceiver device 516 is used by the computer module 501 for communicating to and from a communications network 520, for example connectable via a telephone line 521 or other functional medium. The modem 516 can be used to obtain access to the Internet, other network systems, such as a Local Area Network (LAN) or a Wide Area Network (WAN), and the other personal computer (PC) 522 with which the computer 500 is communicating..

The computer module 501 typically includes at least one processor unit 505, a memory unit 506, for example formed from semiconductor random access memory (RAM) and read only memory (ROM), input/output (I/O) interfaces including a video interface 507, and an I/O interface 513 for the keyboard 502 and mouse 503 and optionally a joystick (not illustrated), and an interface 508 for the modem 516.

A storage device 509 is provided and typically includes a hard disk drive 510 and a floppy disk drive 511. A magnetic tape drive (not illustrated) may also be used. A CD-ROM drive 512 is typically provided as a non-volatile source of data. The components 505 to 513 of the computer module 501, typically communicate via an interconnected bus 504 and in a manner which results in a conventional mode of operation of the computer system 500 known to those in the relevant art. Examples of computers on which the embodiments can be practised include IBM-PC's and compatibles, Sun Sparcstations or alike computer systems evolved therefrom.

Typically, the application program of the embodiment is resident on the hard disk drive 510, and is read and controlled in its execution by the processor 505. Intermediate storage of the program and any data fetched from the network 520 may be accomplished using the semiconductor memory 506, possibly in concert with the hard disk drive 510. In some instances, the application program may be supplied to the user encoded on a CD-ROM or floppy disk and read via the corresponding drive 512 or 511, or alternatively may be read by the user from the PC 522 over the network 520 via the modem device 516.

Still further, the software can also be loaded into the computer system 500 from other computer readable medium including magnetic tape, a ROM or integrated circuit, a magneto-optical disk, a radio or infra-red transmission channel between the computer module 501 and another device, a computer readable card such as a PCMCIA card, and the Internet and Intranets including email transmissions and information recorded on websites and the like. The foregoing is merely exemplary of relevant computer readable mediums. Other computer readable mediums may be practiced without departing from the scope and spirit of the invention.

Industrial Applicability

It is apparent from the above that the embodiment(s) of the invention are applicable to the computer and data processing industries.

The foregoing describes only some embodiments of the present invention, and modifications and/or changes can be made thereto without departing from the scope and spirit of the invention, the embodiments being illustrative and not restrictive.

The Claims defining the invention are as follows:
~~Claims:~~

1. A method of parsing a markup language document comprising syntactic elements, said method comprising, for one of said syntactic elements, the steps of:

identifying a type of the element;

processing the element by determining a hash representation thereof if said type is a first type; and

augmenting an at least partial structural representation of the document using the hash representation if said type is said first type.

2. A method according to claim 1, whereby said parsing is event-based parsing.

3. A method according to claim 1, whereby said hash representation is determined using one of:

a hash algorithm;

a first reference to said hash algorithm dependent upon an associated Universal Reference Indicator;

a second reference to said hash algorithm dependent upon an associated namespace; and

a third reference to said hash algorithm dependent upon an associated Extended Markup Language declaration;

4. A method according to claim 2, whereby said first type is one of:

a structural element;

a definition of said structural element;

a declaration of said structural element; and

a match for said structural element.

5. A method according to claim 4, whereby said structural element is a tag.

6. A method according to claim 2, whereby the hash representation is a unique code for said one syntactic element, said element having less than a first number of characters.

7. A method according to claim 2, whereby the hash representation is not a unique code for said one syntactic element, said element being constrained, to a probability level, in terms of at least one of (i) a number of characters in the element and (ii) a permissible number of permutations of characters in the element.

8. A method according to any one of claims 1 - 7, whereby said code comprises numeric characters.

9. A method according to claim 2, whereby said processing step comprises a sub-step of:

determining an extended hash representation of both (i) said one syntactic element being a first instance of said first type, and (ii) another syntactic element being a second instance of said first type, within which said second instance, said first instance is nested.

10. A method according to any one of claims 1 - 9, whereby the augmenting step is succeeded by a well-formedness checking step against a syntactic rule comprising syntactic elements, said well-formedness checking step comprising sub-steps of:

(a) processing the syntactic rule, said processing comprising, for a syntactic element of the rule, sub-steps of:

(i) identifying a type of said syntactic element of the rule; and

(ii) processing said syntactic element by determining a hash representation thereof if said type is said first type; and

(b) checking said at least partial structural representation of the markup language document against the processed syntactic rule, said checking comprising a sub-step of:

(i) numerically comparing corresponding hashed representations of elements.

11. A method according to claim 10, whereby said numerically comparing step in (b)(i) is succeeded by a further step of:

string-comparing corresponding non-hashed representations of elements not of said first type.

12. A method according to claim 10, whereby said first type is one of:

a structural element;

a definition of said structural element;

a declaration of said structural element; and

a match of said structural element.

13. A method according to claim 12, whereby said structural element is a tag.

14. A method according to claim 10, whereby the well-formedness checking step is one of (a) succeeded by, (b) included in, and (c) replaced by a validation step against a document type definition (DTD), said validation step comprising sub-steps of:

(a) processing the DTD, said processing comprising, for a syntactic element in the DTD, sub-sub-steps of:

(i) identifying a type of said syntactic element of the DTD; and

(ii) processing the syntactic element by determining a hash representation thereof if said type is said first type; and

(b) checking said at least partial structural representation of the markup language document against the DTD, said checking comprising a sub-sub-step of:

(i) numerically comparing corresponding hashed representations of elements.

15. A method according to claim 14, whereby said numerically comparing step in (b)(i) is succeeded by a further step of:

string-comparing corresponding non-hashed representations of elements not of said first type.

16. A method according to claim 14, whereby said first type is one of:

a structural element;

a definition of said structural element;

a declaration of said structural element; and

a match of said structural element.

17. A method according to claim 16, whereby said structural element is a tag.

18. A method of encoding a markup language document comprising syntactic elements, said method comprising, for one of said syntactic elements, steps of:

identifying a type of the syntactic element; and

processing the syntactic element by one of:

(i) determining a hash representation thereof if said type is a first type;

(ii) determining a compressed representation thereof if said type is not a first type; and

(iii) retaining the syntactic element.

19. A method of decoding a markup language document comprising encoded syntactic elements, said method comprising, for one of said encoded syntactic elements, steps of:

identifying a type of the encoded syntactic element;

processing the encoded syntactic element by at least one of:

(i) determining an inverse hash representation thereof if said type is a first type; and

(ii) determining a decompressed representation thereof if said type is not a first type; and

(iii) retaining the encoded syntactic element.

20. An apparatus for parsing a markup language document to form an at least partial structural representation of the document, said apparatus adapted to perform any one of aforementioned methods.

21. An apparatus for one of encoding and decoding a markup language document to form an at least partial structural representation of the document, said apparatus adapted to perform any one of aforementioned methods.

22. A computer program product including a computer readable medium having recorded thereon a computer program for implementing an apparatus for parsing a markup language document to form an at least partial structural representation thereof, said program adapted to perform any one of the aforementioned methods.

23. A method of parsing a markup language document substantially as described herein with reference to the accompanying drawings.

24. An apparatus adapted for parsing a markup language document substantially as described herein with reference to the accompanying drawings.

25. A method for one of encoding and decoding a markup language document substantially as described herein with reference to the accompanying drawings.

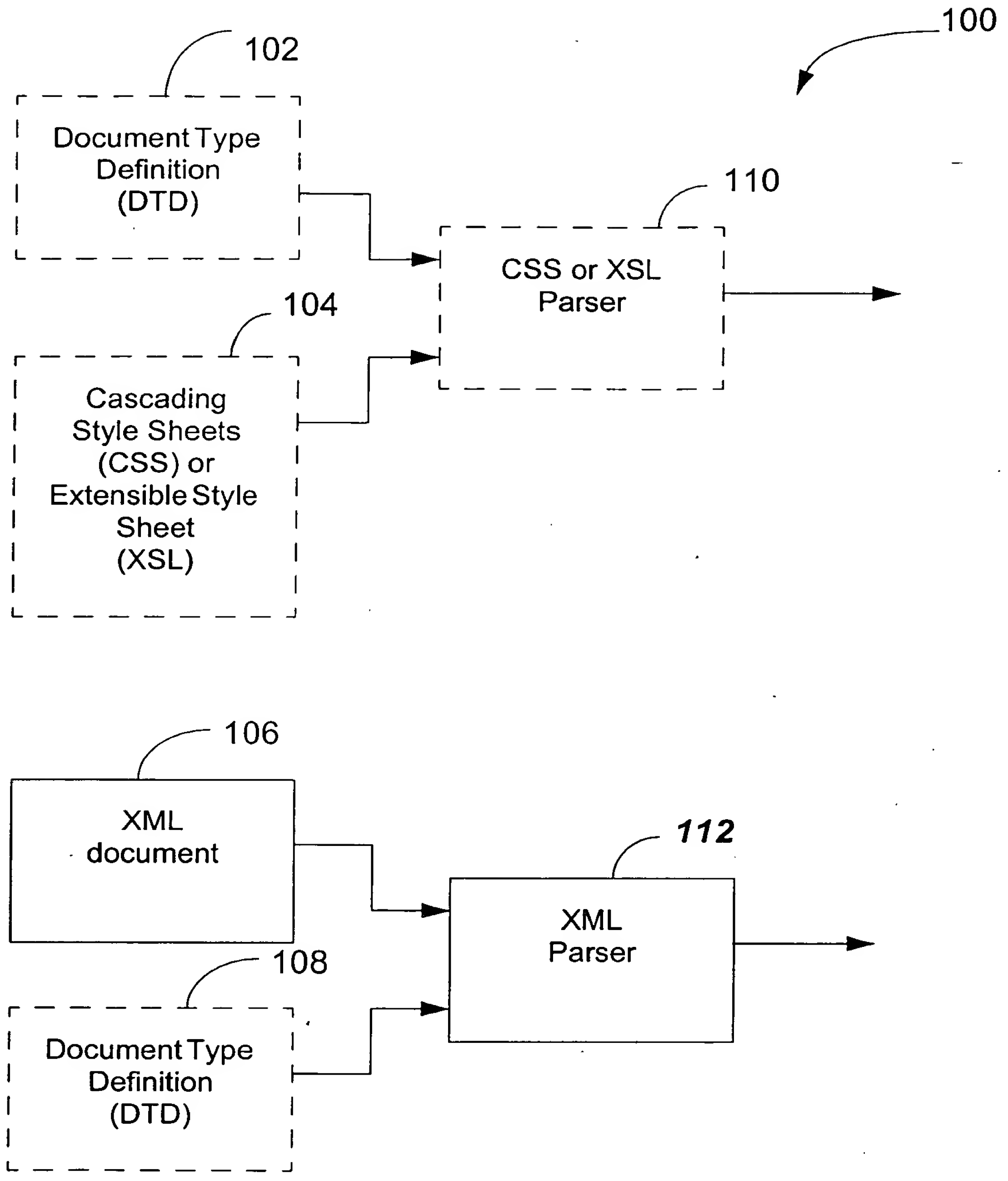
26. An apparatus adapted for one of encoding and decoding a markup language document substantially as described herein with reference to the accompanying drawings.

DATED this Twenty-first Day of June, 2000

Canon Kabushiki Kaisha

Patent Attorneys for the Applicant

SPRUSON & FERGUSON



100
102
Document Type Definition (DTD)
104
Cascading Style Sheets (CSS) or Extensible Style Sheet (XSL)
110
CSS or XSL Parser
106
XML document
112
XML Parser
108
Document Type Definition (DTD)

Fig. 1

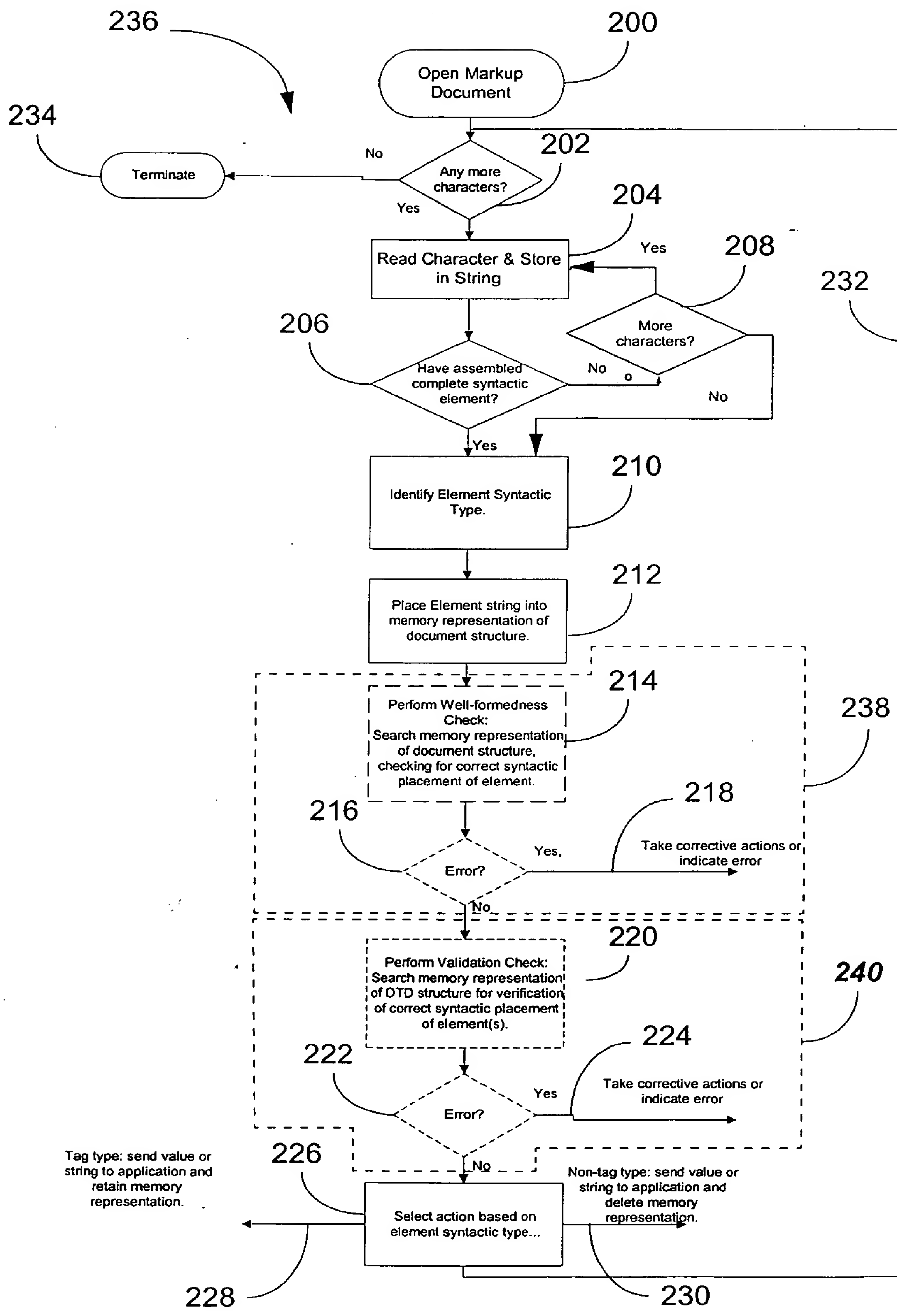

236
200
Open Markup Document
234
Terminate
No
Any more characters?
202
Yes
Read Character & Store in String
204
Yes
208
232
More characters?
206
Have assembled complete syntactic element?
No
No
Yes
Identify Element Syntactic Type.
210
Place Element string into memory representation of document structure.
212
214
Perform Well-formedness Check: Search memory representation of document structure, checking for correct syntactic placement of element.
238
216
218
Yes,
Take corrective actions or indicate error
Error?
No
220
Perform Validation Check: Search memory representation of DTD structure for verification of correct syntactic placement of element(s).
240
224
222
Take corrective actions or indicate error
Yes
Error?
Tag type: send value or string to application and retain memory representation.
226
No
Non-tag type: send value or string to application and delete memory representation.
Select action based on element syntactic type...
228
230

Fig. 2

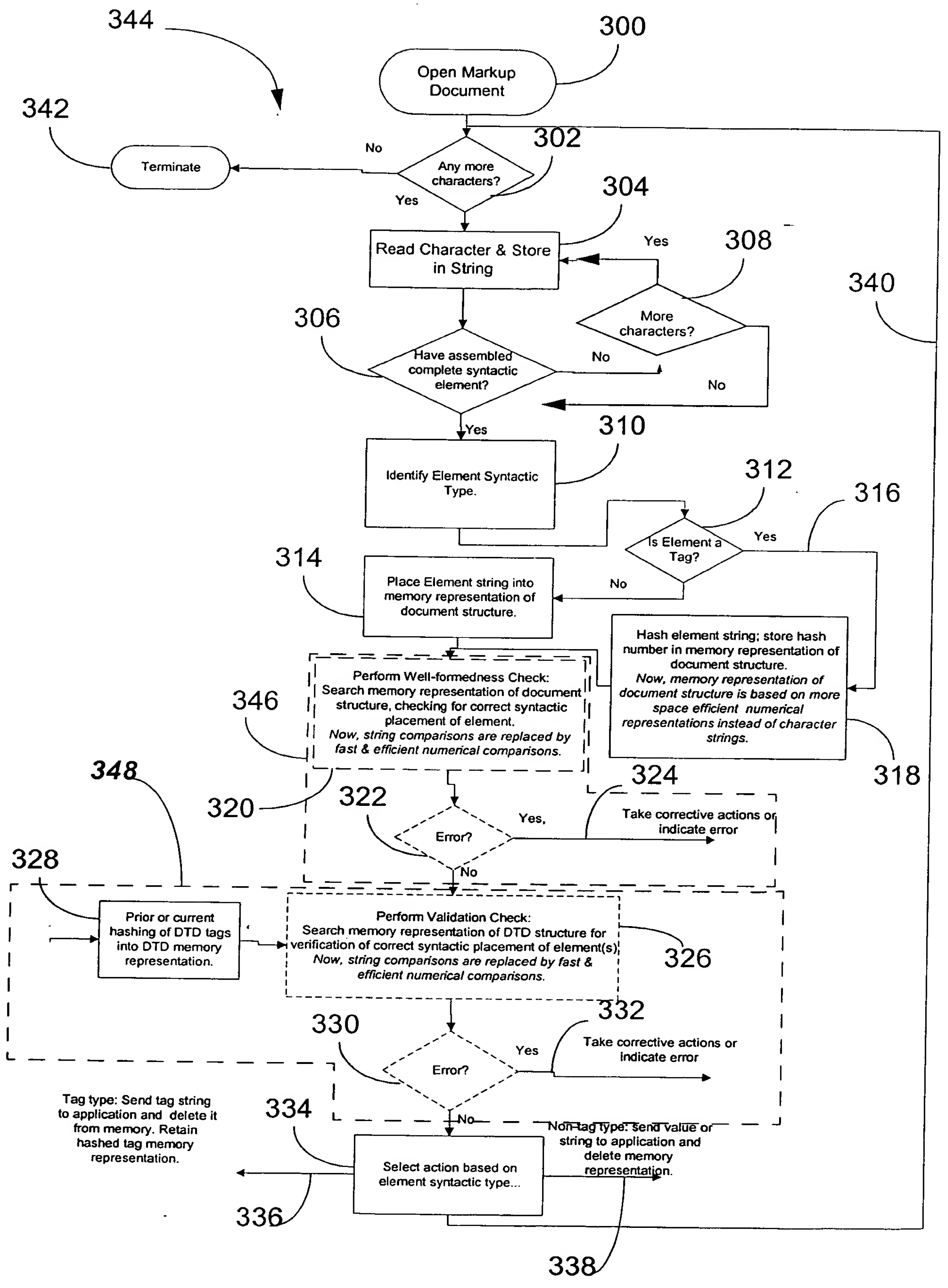

344
300
Open Markup Document
342
Terminate
No
Any more characters?
302
Yes
304
Read Character & Store in String
Yes
308
More characters?
340
306
Have assembled complete syntactic element?
No
No
Yes
310
Identify Element Syntactic Type.
312
316
Is Element a Tag?
Yes
314
Place Element string into memory representation of document structure.
No
Hash element string; store hash number in memory representation of document structure.
Now, memory representation of document structure is based on more space efficient numerical representations instead of character strings.
346
Perform Well-formedness Check:
Search memory representation of document structure, checking for correct syntactic placement of element.
Now, string comparisons are replaced by fast & efficient numerical comparisons.
318
348
324
320
322
Error?
Yes,
Take corrective actions or indicate error
No
328
Prior or current hashing of DTD tags into DTD memory representation.
Perform Validation Check:
Search memory representation of DTD structure for verification of correct syntactic placement of element(s).
Now, string comparisons are replaced by fast & efficient numerical comparisons.
326
330
332
Error?
Yes
Take corrective actions or indicate error
Tag type: Send tag string to application and delete it from memory. Retain hashed tag memory representation.
334
No
Non-tag type: send value or string to application and delete memory representation.
Select action based on element syntactic type...
336
338

Fig. 3

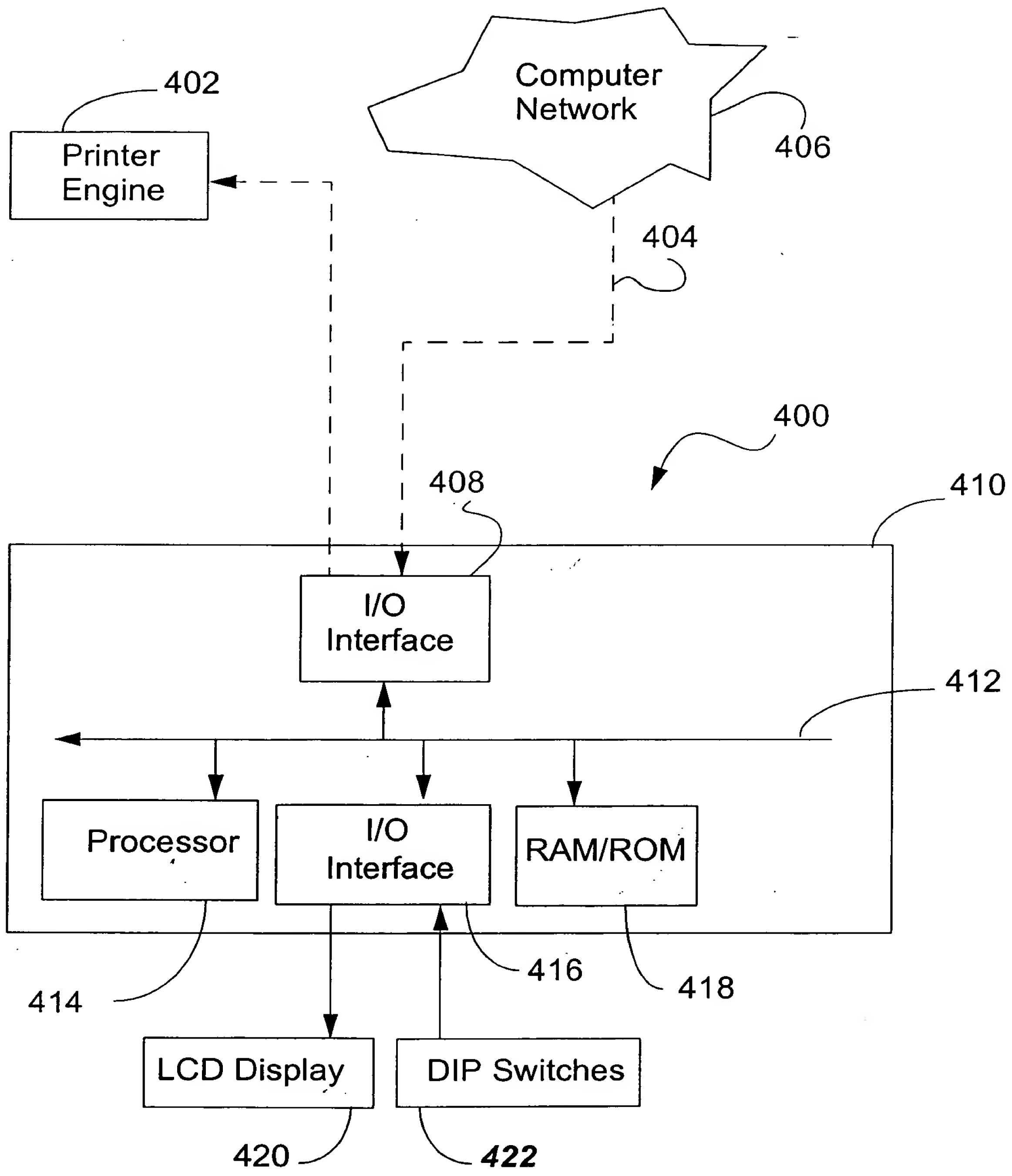

402
Printer Engine
Computer Network
406
404
400
408
410
I/O Interface
412
Processor
I/O Interface
RAM/ROM
414
416
418
LCD Display
DIP Switches
420
422

Fig. 4

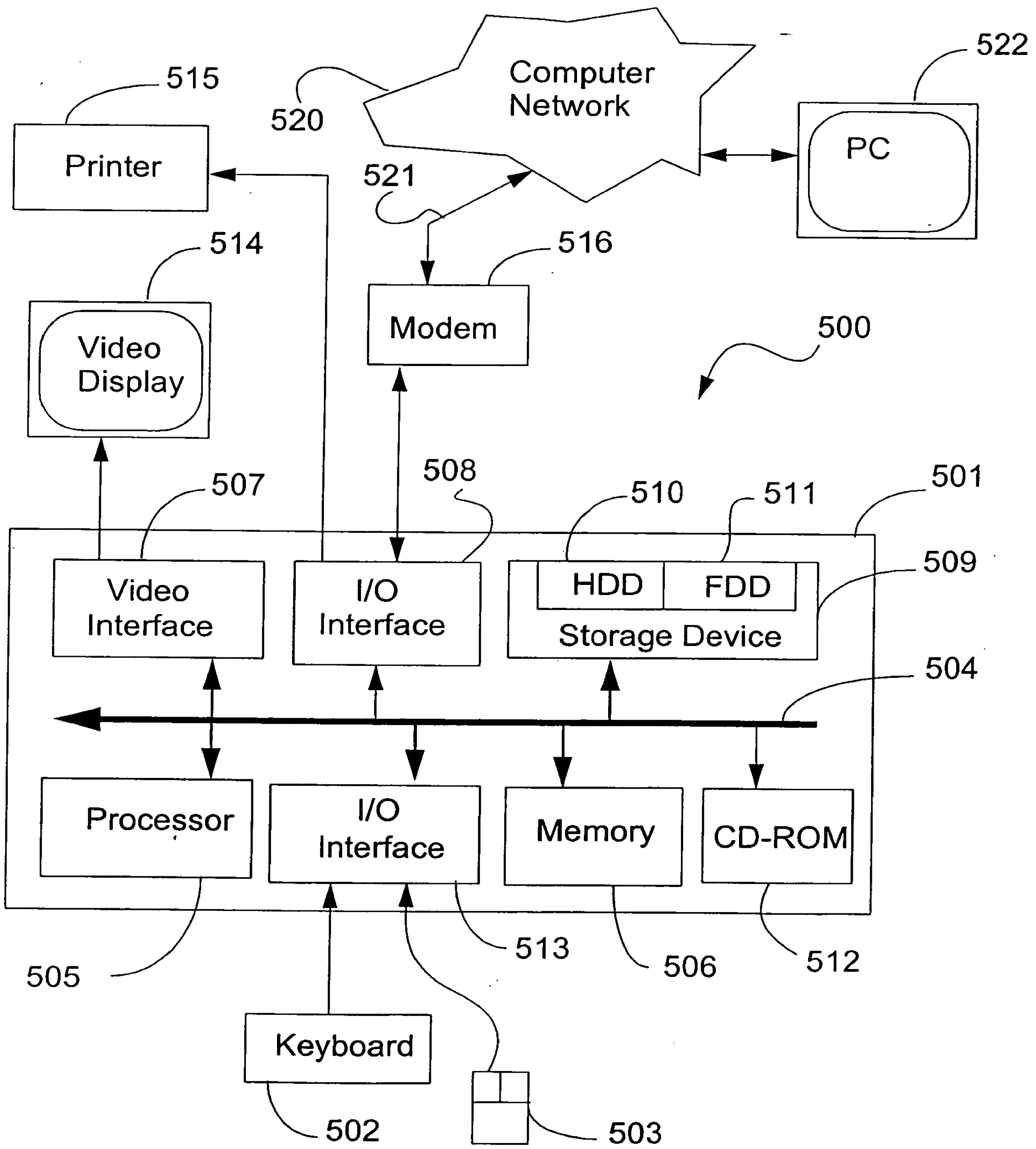

Computer Network
PC
522
515
520
Printer
521
514
516
Modem
500
Video Display
507
508
510
511
501
509
Video Interface
I/O Interface
HDD
FDD
Storage Device
504
Processor
I/O Interface
Memory
CD-ROM
505
513
506
512
Keyboard
502
503

Fig. 5